Multiple-Writer Entry Consistency

Harjinder Sandhu, Tim Brecht, Diego Moscoso
Department of Computer Science, York University,
North York, Canada M3J 1P3.
email: {hsandhu,brecht,diego}@cs.yorku.ca

Abstract *In this paper, we present the design, implementation and evaluation of a new distributed shared memory (DSM) coherence model called multiple-writer entry consistency (MEC). MEC combines the efficient communication mechanisms of Lazy Release Consistency (LRC) with the flexible data management of the Shared Regions [13, 8] and Entry Consistency (EC) models [4]. This is achieved in MEC by decoupling synchronization from coherence (in contrast to the tight coupling of synchronization and coherence present in EC) while retaining the familiar synchronization structure found in Release Consistent (RC) programs. Our experimental evaluation on an 8 processor system shows that using MEC reduces parallel execution times by margins ranging from 5% to 46% in five of the six applications that we study. However, the parallel execution time of the LRC version of the remaining application is lower than the MEC version by 48%. We conclude that offering both page-based and region-based models for coherence within the same system is not only practical but necessary.*

Keywords: Distributed Shared Memory, Coherence Protocol, Memory Consistency

1 Introduction

Within the realm of Distributed Shared Memory (DSM) systems, several divergent streams of models have emerged. Among the most common are those that rely on pages as the underlying abstraction for data management and use Release Consistency (RC) as the synchronization model presented to the user [5, 9], and those that use user-defined *shared regions* as the underlying abstraction for data management coupled with a synchronization model called Entry Consistency (EC) that explicitly binds synchronization primitives with the data they protect [4, 8, 12]. Each of these models has their limitations. In RC-based systems, the reliance on pages as the unit of data management often causes these systems to transmit more data than necessary in order to maintain coherence due to the effects of false sharing, or to transmit data using more messages than necessary when shared data structures span multiple pages. Sophisticated coherence protocols such as *lazy release consistency* (LRC) reduce the impact of false sharing but cannot eliminate it entirely [9]. In EC-based systems, the problems that stem from the mismatch between the system's view of sharing and the granularity of sharing within the application are avoided entirely by managing data at user-defined granularity. However, the EC synchronization model is more complex to program than RC and is often avoided as a result.

Ideally, a system should integrate both page-based and region-based protocols in a way that permits the programmer to begin with the simpler RC-based model, and then add EC annotations for just those data structures that are better suited to region-based rather than page-based data management. Unfortunately, RC and EC are difficult to integrate in this way due to the differences in their respective synchronization models. Adding EC annotations in an existing RC program requires modifying the program's synchronization structure and it may result in a program that con-

tains too much synchronization. The study of LRC versus EC by Adve *et al.* [1], for instance, points out that their EC-based applications often suffered from this problem of over-synchronization.

In this paper, we present a new model called Multiple-Writer Entry Consistency (MEC). MEC manages data at a granularity defined by the user but, unlike EC, it decouples the primitives used for synchronization from those used for coherence. This decoupling of synchronization from coherence has two effects. The first is that it permits the efficient integration of both page-based and region-based protocols for coherence within the same RC framework. For synchronization, MEC retains the familiar *acquire* and *release* primitives of RC. Consequently, users can write programs for the simpler RC model, using an efficient page-based protocol such as LRC, and then, without changing the synchronization structure of the program, selectively add MEC annotations to those data structures that may benefit from region-based data management.

The second effect of decoupling synchronization and coherence in MEC is that user-defined shared regions can be concurrently modified by different processors. This presents greater flexibility than the single-writer multiple-reader behavior enforced by EC on regions defined by the user. Consider, for example, an application with interspersed fine-grained sharing among the elements of a large data structure. Under typical EC implementations, either each element would have to be bound to a separate synchronization object and guarded individually, or else the entire structure would have to be bound to a single synchronization object and guarded in a way that permits only one processor to modify it at a time. The former partitioning would be too fine-grained to be efficient in a DSM environment, while the latter solution would unnecessarily restrict concurrency.

We have implemented the MEC protocol within the Treadmarks distributed shared memory system (which itself implements page-based LRC). We present the results of an evaluation that compares the performance of LRC and MEC for six applications. The results of this investigation show that five of the six applications obtain better performance using MEC than LRC, with differences ranging from 5% to 46%. In the remaining application, however, there is a significant degradation (48%) using MEC. These results argue strongly in favor of using an integrated page-based and region-based solution as a platform for distributed shared memory.

An overview of this paper is as follows. Section 2 briefly describes the background to this work with a description of the RC, LRC, and EC models. Section 3 describes the MEC model and its implementation. Section 4 presents the results of a performance evaluation of LRC versus MEC, and is followed in Section 5 with a discussion of how these results relate to other work, including earlier work comparing LRC and EC. Section 6 presents the conclusions of this paper.

2 Background

Memory consistency models present a set of programming constraints to the user and guarantee correctness only for programs that obey those constraints. In the Release Consistency (RC) model [7] these constraints are (a) that only system recognized synchronization operations are used, (b) that all synchronization operations are labeled as *acquire* or *release* operations, and (c) that there are no data races in the program. Under RC, a processor may delay the propagation of coherence related information to other processors until it arrives at a *release* point in the program. A release point in the program indicates that the process is exiting a critical section, and that other processors must now be made aware of modifications to shared data within this critical section. A lazy implementation of release consistency (LRC) [9], however, takes advantage of the fact that a processor that wishes to use the data computed by another processor must first acquire a lock from the processor that has modified

that data. Therefore, using LRC the propagation of coherence related information can be delayed until the acquire operation by another processor, rather than immediately upon a release operation.

DSM implementations using RC mitigate the impact of false sharing on pages both by delaying the propagation of coherence information and by permitting multiple processors to concurrently modify different portions of the same page. Munin, among the first DSM systems to use RC, introduced the notion of *twinning* and *diffing* to consolidate concurrent changes to a page from different processors [5]. Prior to modifying a page, a processor creates a twin (a local copy) of the page. Later, upon arrival at a release point, it compares the modified page to its unmodified twin and transmits to other processors only those portions of the page that have changed. Treadmarks' implementation of LRC delays the creation and transmition of *diff*'s until modifications to the page are requested by another processor, thereby further reducing the number of messages required to maintain coherence.

Entry Consistency (EC) [4], and other models like it, such as the Shared Regions (SR) model [8, 13], avoid the problem of false sharing among virtual memory pages by managing data at a granularity defined by the user.[1] In SR, a *shared region* is a user-defined entity describing the granularity at which data is shared within the application. According to both EC and SR, each shared region is explicitly bound in the program to a token that is used for synchronization. In SR, the primitives *readaccess*, *writeaccess*, *readdone*, and *writedone* are used to guard access to a shared region. In addition to enforcing synchronization, the underlying system checks the state of the region when these primitives are invoked and, when necessary, initiates coherence actions to keep the region consistent. Each user-defined shared region follows a single-writer multiple-reader synchronization protocol. The EC or SR models have been used in a number of systems, including Midway [4], CRL [8], DiSom [12], Amber [6], and Hurricane [13].

[1] For the purposes of discussion, we borrow terminology from both the EC and SR models; semantically, these models are equivalent.

3 Multiple-Writer Entry Consistency

The Multiple Writer Entry Consistency (MEC) model described in this paper combines elements of both RC and EC in the interface that is presented to the user, and is akin to LRC in its implementation. This section describes the MEC model and its implementation.

3.1 Model

As in the Shared Regions (SR) model [13], we use the term *shared region* in MEC to denote a user-defined region of memory that is treated as a single unit for the purposes of coherence enforcement. Once a set of shared regions has been defined within the MEC framework, a set of operations, *readaccess* and *writeaccess*, are used in conjunction with a particular shared region to indicate when references to that shared region occur, and whether these references also modify that region. We refer to these operations generically as *access* operations.

From the programmers perspective, the main differences between MEC and the EC and SR models are twofold. First, the access operations in MEC enforce coherence but are non-synchronizing. For synchronization, MEC retains the familiar *acquire* and *release* operations of RC, and integrates the coherence enforced through the *access* operations with the synchronization performed by the *acquire/release* operations. The second difference between SR and MEC is that, in MEC, although the access operations are used prior to a series of references to a particular region, there is no need for corresponding operations to indicate the completion of this series of references to the region.

In MEC, regions defined by the user can be modified concurrently by different processors. Access operations are used to determine when

to consolidate changes to a region. An *access* operation by processor P_i ensures that all modifications to region R *preceding* the *access* operation are seen by P_i. The use of the term *preceding* in MEC is equivalent to the *happened-before-1* relationship defined by Adve *et al.* [2]. That is, x can be said to *precede* y if: (a) x and y are on the same processor and x occurs before y in the order defined by the program, or (b) x is a release operation on one processor and y is an acquire operation on another processor, and y returns the value written by x. This relationship is transitive, so if x precedes y and y precedes z, then x precedes z in this partial order. Thus, modifications to R by processor P_i will be seen by processor P_j only when there is a transitive chain such that the modifications by P_i precede an *access* operation by P_j.

3.2 Implementation

Our implementation integrates MEC into the Treadmarks implementation of LRC [9]. In the standard Treadmarks implementation, a large shared data space is created in the initialization of the system, and all shared data is subsequently allocated within this space. A page table is used to keep track of the state of each page in this shared data area. The execution of each processor is divided into time intervals (where synchronization operations mark the beginning and end of an interval) and each processor maintains a list of the intervals it has seen from each of the other processors. On an acquire operation, a processor transmits its current interval time stamp and receives in return a list of intervals it has not seen and *write notices* containing the set of pages that were modified in that interval.

In our implementation of MEC, the shared data area is partitioned into two sections, a page-based area containing K virtual memory pages, and a shared region-based area containing the remaining space. The page table is converted into a page-region table, partitioned so that the first K entries keep track of pages in the page area, and the remaining entries keep track of user-defined regions in the shared region area. Shared data may be allocated in either area from within the program. Data allocated in the page-based area is managed using LRC in exactly the same way as in the unmodified Treadmarks system, using virtual memory protection to intercept faulting read/write references to pages and initiate coherence actions. For data in the shared regions area, page faults do not occur. Instead, *access* annotations in the program are used to determine when coherence actions are needed for a particular region.

In the integrated LRC/MEC implementation, both pages and shared regions are referred to by their index in the page-region table. On an acquire operation, a processor transmits its vector time stamp to the processor currently holding the lock, just as it does by default in LRC, but it receives in response *write notices* containing a list of both pages and regions that have been modified in intervals not seen by the acquiring processor. In MEC, a region is considered modified in interval q by P_i if P_i issued a *writeaccess* on that region in interval q.

The *diff* and *interval* management routines are identical for both page-based LRC and region-based MEC except for the granularity at which these actions are carried out. In our implementation, changes were made to the original Treadmarks *diff* management routines to support arbitrary regions of data, but the interval management routines were not modified from the originals. We also converted the Treadmarks communication layer from UDP to TCP to facilitate the transmition of large data regions in MEC. Comparisons between LRC in the original (unmodified) Treadmarks system and our modified version showed that neither of these changes had a significant impact on performance.

4 Performance Evaluation

In order to assess the performance of MEC as well as the value of integrating both MEC and LRC within the same system, we compared

the performance of six application kernels executed first using LRC, and then with key shared data structures identified as shared regions in order to use MEC. These applications are: Blocked Matrix Multiplication (MM), Successive Over-Relaxation (SOR), Blocked-Contiguous LU-Decomposition (LU), Traveling Salesman Problem (TSP), Integer Sort (IS), and Floyd-Warshall (FLOYD). TSP, SOR, and IS are from the suite of applications used in earlier TreadMarks studies [9, 11, 1], LU is from the Splash-2 benchmark suite [15], and MM and FLOYD were written locally.

All experiments were conducted on an 8 node cluster of workstations connected together by a Fore ATM switch. The workstations are IBM RISC System/6000s each with a 133 MHz PowerPC 604 processor, 16 Kb L1 instruction cache, 16 Kb L1 data cache, a 4-way associative 512 Kb L2 cache, and 96 Mb of memory. Virtual memory pages are 4 Kb in size. Table 1 shows the problem sizes, execution times, and speedups of these applications running on this workstation cluster.

Figure 1(a) shows the normalized parallel execution times of the applications used in this study. Execution times are divided into four components: (1) sync - synchronization time, (2) traps - time spent handling traps (protection traps to pages in LRC, *access* traps in MEC), which includes time spent fetching diffs and creating twins, (3) remote - time spent handling remote requests initiated from other processors, including time spent creating diffs on demand, (4) base - the time remaining, once sync, traps, and remote costs are accounted for. The base time is largely spent performing computation in the application, although because trap and remote costs begin timing after asynchronously entering the signal handler, the base time will also include the cost of entering the signal handler. The accompanying graphs in Figure 1 show the number of messages communicated and the number of bytes transferred relative to one processor.

In the MEC version of these applications, shared data structures were partitioned into shared regions in what seemed to be the most obvious and natural fashion for each application. For the problem sizes used, MEC execution times are lower than LRC in MM, SOR, LU, FLOYD, and IS, by margins ranging from 5% in IS to 46% in SOR (Figure 1(a)). In TSP, MEC performance is significantly worse than that of MEC, by a factor of about two. The two most important factors influencing the relative performance of these applications is the number of messages and the number of bytes that are transmitted between processors, shown in Figure 1(b) and Figure 1(c) respectively. In all cases except Floyd and IS, the number of messages transmitted using MEC is significantly lower than using LRC. The number of bytes transmitted between processors is, on the other hand, about the same for both models in three cases, MM, IS, and SOR, and higher in MEC than LRC in two cases, LU and TSP.

In Floyd, the greater number of bytes transmitted in LRC versus MEC is due to false sharing present in the LRC implementation. In most of the other applications, careful data alignment minimizes false sharing effects under LRC. However, in two applications, LU and TSP, the number of bytes transmitted is substantially higher in MEC than in LRC. This is because, in these cases, entire regions are being modified by one processor while only a portion of that region is used by another processor. In MEC when a region is accessed all modified portions of the region are transmitted, while in LRC only those pages that are faulted on are sent. This suggests that the MEC version of these applications might be improved by declaring these regions at a finer granularity. However, this would increase the number of messages while decreasing the number of bytes transmitted.

In cases where these regions are significantly larger than the size of a page (MM, SOR, LU, IS, TSP), the MEC algorithm requires many fewer messages to obtain the same data. For instance, in MM, where each region is defined to contain only those portions of each of the three matrices required by a particular processor, less than 8 messages are required to fetch all of the data needed by that processor. In the

program	problem size	1 proc	8 proc time		speedup	
		time	LRC	MEC	LRC	MEC
MM	1536x1536 (int) matrices , block size 32	257.7	58.1	50.1	4.4	5.1
SOR	2000x2000 (float) matrix, 100 iterations	108.5	34.9	18.8	3.1	5.8
LU	1536x1536 (double) matrix, block size 64	97.9	35.8	21.2	2.7	4.6
TSP	19 city tour	56.2	23.3	44.5	2.4	1.3
FLOYD	512 node graph	61.9	13.8	12.5	4.5	5.0
IS	2^{22} keys, bucket size 2^9, 10 iterations	24.9	4.0	3.9	6.2	6.3

Table 1: Problem sizes, execution times (in seconds), and speedups on 8 processors.

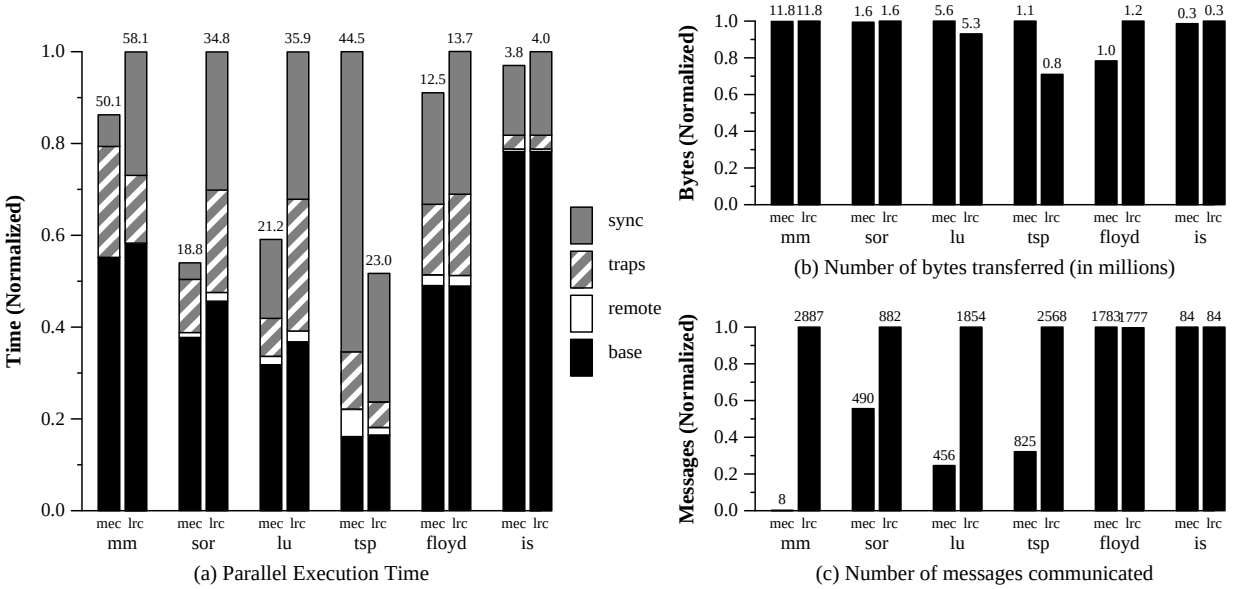

Figure 1: Eight processor performance of applications under LRC and MEC.

LRC version of MM, the number of messages required is much higher, about 2887, as pages are fetched one at a time. In SOR and LU, regions span about two and four pages respectively, so the number of messages is reduced by factors of approximately two and four. In TSP, a single region spans about one hundred pages, but the region is not usually needed in its entirety by a particular processor. Consequently, although the number of messages is reduced by a factor of 4, the number of bytes transmitted is also significantly higher. This results in the worse performance of MEC in this one case.

The performance of MEC and LRC in these applications highlights the need for providing both page-based and region-based protocols within the same system. MEC requires more effort to program, and an improvement of about 5%, as in Floyd, is not likely to justify this effort. Further, defining regions poorly (as in TSP) can cause a significant degradation in performance through false sharing within a single region. On the other hand, MEC does provide substantial improvements in performance for four applications, from 12% to 46%. In these cases, the additional effort of using MEC rather than LRC would appear to be justified.

5 Related Work

Adve *et al.* [1] conducted a performance comparison of lazy release consistency [7] and entry consistency [4] and conclude that there is no clear winner in terms of performance. They also point out that the performance of EC is hurt by extra synchronization and lock rebinding because all accesses to regions of shared memory may only occur after acquiring the corresponding lock. In contrast, the MEC model proposed in this paper uses no extra synchronization and requires no lock rebinding, yet retains the ability to manage data at region granularity. Further, in the three applications (SOR, TSP, and IS) that are common to both our study and the study by Adve *et al.*, our results show that variations in problem size have a significant impact on the difference between page-based and region-based data management. In SOR, for example, for the problem size used by Adve *et al.* (1000x1000 matrices), there is no significant difference between LRC and EC in their study and no significant difference in performance between LRC and MEC in our study. However, larger problem sizes in SOR are shown to favor MEC over LRC in our study.

The use of larger virtual memory pages for obtaining data aggregating effects in LRC has been previously considered by Amze *et al.* [3]. While larger virtual memory pages improves LRC performance for many applications, its effectiveness is limited by the increase in false sharing in others. On the other hand, MEC is capable of simultaneously eliminating false sharing along page boundaries and achieving much larger data aggregation, on the order of megabytes in some of the MEC applications we examine, in contrast with the 16 Kb pages used in the study by Amze *et al.*.

Neves *et al.* [12] conduct a comparison of TreadMarks (LRC) and their system, called DiSOM, which implements EC. Their results show that EC can send significantly fewer messages and less data and thus obtain substantial reductions in execution times compared to LRC. The study by Neves *et al.* differs from our own in several respects, aside from the differences between MEC and EC already noted. The DiSOM system uses an object-oriented framework to avoid write trapping and exercise fine-grain control over communication, and their comparison between LRC in Treadmarks and EC in DiSOM is between two different DSM implementations that may differ in other respects as well. In contrast, our study compares MEC and LRC, both implemented within the Treadmarks framework.

6 Conclusions

In this paper, we have presented a new model called Multiple-Writer Entry Consistency (MEC). The value of MEC is in its use of an RC-based synchronization structure while managing data at a granularity defined by the user. Thus, MEC permits the efficient integration of both page-based and region-based coherence protocols within the same RC framework. In this integrated environment, a program can be written initially for the RC model using page-based data management, and then MEC annotations added for those data structures which may obtain a performance advantage from region-based data management. MEC also presents a significant amount of flexibility in the way shared regions are defined by the user, by allowing a single region to be modified concurrently by different processors.

Our experimental evaluation of the performance of MEC and LRC shows that MEC improves execution times in five of the six applications that we study, by margins ranging from 5% to 46% for the default problem sizes, while leading to a degradation in performance of 48% in one case. These results suggest that integrating both page-based and region-based coherence protocols into one DSM framework is both practical and necessary.

References

[1] S.V. Adve, A.L. Cox, S. Dwarkadas, R. Rajamony and W. Zwaenepoel, "A Comparison of Entry Consistency and Lazy Release

Consistency Implementations", Proceedings of the 2nd International Symposium on High-Performance Computer Architecture, pp. 26-37, February, 1996.

[2] S.V. Adve and M.D. Hill, "Weak Ordering – A New Definition", Proceedings of the 17th Annual International Symposium on Computer Architecture, pp. 2-14, May 1990.

[3] C. Amza, A.L. Cox, K. Rajamani, and W. Zwaenepoel, "Tradeoffs between False Sharing and Aggregation in Software Distributed Shared Memory", Proceedings of the Sixth Conference on Principles and Practice of Parallel Programming, pp. 90-99, June 1997.

[4] B. Bershad, M. Zekauskas and W. Sawdon, "The Midway Distributed Shared Memory System", Proceedings of COMPCOM '93, pp. 528-537, February, 1993.

[5] J. Carter, J. Bennett and W. Zwaenepoel, "Implementation and Performance of Munin", Proceedings of the 13th Symposium on Operating Systems Principles, pp. 152-164, October, 1991.

[6] M. Feeley and H. Levy, "Distributed Shared Memory with Versioned Objects", Proceedings of the Conference on Object-Oriented Programming Systems Languages, and Applications, October, 1992.

[7] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, J. Hennessy, "Memory Consistency and Event Ordering in Scalable Shared Memory Multiprocessors", Proceedings of the 17th Annual Symposium on Computer Architecture, pp. 15-26, May, 1990.

[8] K. Johnson, F. Kaashoek and D. Wallach, "CRL: High-Performance All Software Distributed Shared Memory", Proceedings of the 15th Symposium on Operating Systems Principles, pp. 213-228, December, 1995.

[9] P. Keleher, A. Cox, S. Dwarkadas and W. Zwaenepoel, "TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems", Proceedings of the Winter 1995 USENIX Conference, pp. 115-131, 1994.

[10] K. Li and P. Hudak, "Memory Coherence in Shared Virtual Memory Systems", ACM Transactions on Computer Systems, Vol. 7, No. 4, pp. 321-359, November, 1989.

[11] H. Lu, S. Dwarkadas, A.L. Cox, and W. Zwaenepoel, "Message Passing Versus Distributed Shared Memory on Networks of Workstations", Proceedings of Supercomputing '95, December, 1995.

[12] N. Neves, M. Castro, and P. Guedes, "A Checkpoint Protocol for an Entry Consistent Shared Memory System", Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing August, 1994.

[13] H. Sandhu, B. Gamsa and S. Zhou, "The Shared Regions Approach to Software Cache Coherence on Multiprocessors", Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 229-238, May, 1993.

[14] I. Schoinas, B. Falsafi, A.R. Lebek, S.K. Reinhardt, J.R. Larus, and D.A. Wood, "Fine-grain Access Control for Distributed Shared Memory", Proceedings of the 6th Symposium on Architectural Support for Programming Languages and Operating Systems, pp. 297-306, October, 1994.

[15] J.P. Singh, W.-D. Weber and A. Gupta, "SPLASH: Stanford Parallel Applications for Shared-Memory", Computer Architecture News, Vol. 20, No. 1, pp. 5-44, March, 1992.

[16] Z.G. Vranesic, M. Stumm, D.M. Lewis and R. White, "Hector - A Hierarchically Structured Shared-Memory Multiprocessor", IEEE Computer, Vol. 24, No. 1, pp. 72-29, January, 1991.

[17] M.J. Zekauskas, W.A. Sawdon, and B.N. Bershad, "Software Write Detection for Distributed Shared Memory", Proceedings of the First Symposium on Operating System Design and Implementation, pp. 87-100, November, 1994.